How Programmers Read Regular Code: A Controlled Experiment Using Eye Tracking

Ahmad Jbara[1,2] Dror G. Feitelson[2]
[1]School of Mathematics and Computer Science
Netanya Academic College, 42100 Netanya, Israel
[2]School of Computer Science and Engineering
The Hebrew University of Jerusalem, 91904 Jerusalem, Israel

***Abstract*—Regular code, which includes repetitions of the same basic pattern, has been shown to have an effect on code comprehension: a regular function can be just as easy to comprehend as an irregular one with the same functionality, despite being longer and including more control constructs. It has been speculated that this effect is due to leveraging the understanding of the first instances to ease the understanding of repeated instances of the pattern. To verify and quantify this effect, we use eye tracking to measure the time and effort spent reading and understanding regular code. The results are that time and effort invested in the initial code segments are indeed much larger than those spent on the later ones, and the decay in effort can be modeled by an exponential or cubic model. This shows that syntactic code complexity metrics (such as LOC and MCC) need to be made context-sensitive, e.g. by giving reduced weight to repeated segments according to their place in the sequence.**

I. Introduction

Although there is a general agreement on the importance of code complexity metrics, there is little agreement on specific metrics and in particular their accuracy [26]. Syntactic metrics like lines of code (LOC) and McCabe's cyclomatic complexity (MCC) are commonly used mainly because they are simple. These metrics are additive and myopic: they simply count source code elements without considering their context and type. Therefore, they do not necessarily reflect the "effective complexity" of source code. In particular, they lead to inflated measurements of well-structured long functions that are actually reasonably simple to comprehend [11].

In previous work [11], [10], we introduced *regularity* as a new factor that questions the additivity of the classical syntactic metrics. Regularity is the repetition of code segments (patterns), where instances of these patterns are usually successive. Figures 4 and 5 show examples of regular code (the repeated instances are indicated by rectangles).

Regular code is generally longer than its non-regular alternative, and if measured by metrics like MCC it is also more complex, as there is a strong correlation between LOC and MCC. However, our experiments showed that long "complex" regular code is not harder to comprehend than the non-regular alternative which is shorter normal code. The speculation was that regularity helps because repeated instances are easier once the initial ones are understood [10].

To investigate this idea, we conducted a controlled experiment that uses eye tracking to explore how programmers read regular code, and to quantitatively measure the time and effort invested in the successive repetitions of such a code. The results indeed show that the time and effort are reduced as subjects progress from one repeated instance of a pattern to the next. This reduction can be modeled by an exponential or a cubic function.

The consequence is that additive syntax-based metrics like LOC or MCC may be misleading, because repeated instances contribute less to complexity and comprehension effort. This observation was made already by Weyuker in the context of her famed work on desirable properties of code complexity metrics [29], where she writes "Consider the program body P; P (that is, the same set of statements repeated twice). Would it take twice as much time to implement or understand P; P as P? Probably not." Our results enable us to take an additional step, and suggest a specific weighting function which can be applied to repeated code segments so as to reflect their reduced effect. This adds a degree of context sensitivity to previously oblivious syntactic metrics.

II. Motivation and Research Questions

While a large number of metrics for measuring code complexity have been proposed, no one of them is capable of fully reflecting the complexity of source code [4], [16]. In previous work we have suggested *regularity* as an additional factor that affects code comprehension, especially in long functions, and provided experimental evidences for its significance [11], [10]. Specifically, we conducted several experiments where developers were required to understand functions and to perform maintenance tasks on them, where different subjects were actually working on different versions of the same function. Thus we could evaluate the relationship between performance and the style in which the function was coded.

To quantify the level of regularity of the different versions, we use an operational definition that is based on compression. We have systematically investigated different compression schemes and code preprocessing levels [9], [11], and found that different combinations yield different results. The combination that gave the best correlation with perceived complexity was to strip the code down to a skeleton of keywords and formatting, and use the *gzip* compression routine. Regularity is quantified by the compression ratio.

TABLE I
ATTRIBUTES OF THE TWO VERSIONS OF THE PROGRAMS USED IN THE EXPERIMENT. THE REG. COLUMN REPRESENTS COMPRESSION RATIO.

	Regular version			Non-regular version			
Program	LOC	MCC	Reg.	LOC	MCC	Reg.	Description
Median	53	18	79.3%	34	13	60.7%	Find medians of all 3×3 neighborhoods
Diamond	46	17	82.8%	26	14	43.8%	Find max Manhattan-radius around point with all same value

Regular functions by definition contain repetitive code segments that, at least to some extent, come one directly after another. This implies that understanding one of these segments would help understanding subsequent ones. Based on this we argue that the cognitive effort needed for the second segment is lesser than that for the first, and as the developer proceeds in the sequence of repetitive segments the effort needed becomes smaller. After several segments it may be expected that the additional effort would even be negligible.

The purpose of this work is twofold. First is to replicate our previous study regarding where we compared the performance of subjects when maintaining regular programs versus their non-regular counterpart. The second purpose is to study the way developers investigate regular code, and whether their efforts in repetitive segments are equal. If the efforts are not the same we want to find a model that reflects the relation between the serial location of the segment and the amount of effort needed to comprehend it. If developers need less effort to understand repeated segments that they already encountered then our model would be a good context-dependent weighting factor for metrics that consider all segments using the same mechanism and hence yield exaggerated measures. An example of very long functions that some metrics classify as very hard while humans classify as simple and well structured was presented in [11].

The research questions this paper addresses are:

- Do developers follow any pattern when they are required to comprehend regular code? In particular, are their efforts equally divided among regular segments?
- Assuming there is a pattern that governs their understanding behaviour, which model might fit and describe it?
- In terms of correctness and completion time, are the results consistent with those of our previous work [10]?

III. METHODOLOGICAL APPROACH

A. Test Programs

We use two programs from the image processing domain (Table I). Each program has two versions: regular and non-regular. The specifications of the programs used are: *finding the medians of all 3×3 neighborhoods* and *finding the maximal Manhattan-radius around a point with all same value*. The programs were taken from our previous work [10], and they meet the following design criteria:

- Realistic programs of known domain.
- Reasonable regular and non regular implementations of the same specification.
- Non trivial specifications

We could use one program with its two implementations, but we prefer two programs to avoid program-specific conclusions.

B. Eye-tracking Apparatus

We use the *Eye Tribe* eye tracker (www.theeyetribe.com) in this work. The device uses a camera and an infrared LED. It operates at a sampling rate of 60Hz, latency less than 20ms at 60Hz mode, and accuracy of 0.5°–1°. The device supports 9, 12, or 16 points for the calibration process. We used 9 points mode. The screen resolution was set to 1280 by 1024.

The *Eye Tribe* is a remote eye tracker and as such it provides the subjects a non-intrusive work environment which is essential for reliable measurements. Furthermore, the device allows head movements during the real experiment but not while calibrating.

To analyze the tracking data we use *OGAMA* (www.ogama.net). It is an open source software designed for analyzing eye and mouse movements. *OGAMA* supports many commercial eye trackers like *Tobii*. In its last version (4.5) support for the *Eye Tribe* has been added. This builtin support makes the process easier and saves the import of the data between systems.

C. Task Design

Basically, we adopted the programs and the task of experiment 2 from our previous work [10] with one difference. In our previous work, each subject sequentially performed the same task (*understanding what does a program do*) for the regular version of one program and the non-regular version of the other. In this work we follow a between-subject design where each subject performs the task on one version only. This design decision has been taken on the basis of a pilot study where subjects claimed that performing two programs is hard especially when you have to keep your gazes within the screen for a long time [27].

In addition to answering the comprehension question *what does the program do*, the subjects were asked to evaluate the difficulty of the code on a 5-point scale, and state the reasons for their evaluation.

A post-experiment question was presented to each participant regarding the way they approach the programs, with the goal of understanding how their effort was distributed in the code and why. Retrospectively, it turned out that this post-experiment question was important as there were cases where the eye tracking data did not fit the participant's opinion.

D. Grading Solutions for Correctness

In grading the solutions of the subjects we followed [14], [3], [23]. In particular, we adopted a multi-pass approach where three evaluators were involved. Initially, the first author evaluated the answers according to a personal scale. In the

TABLE II
AVERAGE GRADES OF THE PARTICIPANTS IN THE DIFFERENT GROUPS.

Style	All courses	Programming courses
Regular (median)	84.0±7.9	86.5±11.1
Regular (diamond)	86.6±9.0	87.0±9.8
	85.1±8.1	86.7±10.0
Non-regular (median)	82.2±11.0	83.2±9.9
Non-regular (diamond)	85.6±8.1	86.2±7.7
	84.1±9.0	84.8±8.3

TABLE III
ACCURACY LEVELS OF THE CALIBRATION PROCESS AND HOW MANY SUBJECTS FALL INTO EACH OF THESE LEVELS.

Level	Accuracy	subjects
Perfect	$< 0.5°$	12
Good	$< 0.7°$	5
Moderate	$< 1.0°$	2
Poor	$< 1.5°$	0
Re-calibrate	bad	1

second pass another colleague evaluated the answers. However, in a few cases there were large gaps between the two evaluations. To resolve this, the second author made a third pass on these cases.

The final grade for each of the cases was computed as the average of the three evaluations when these were close enough ($\leq$10 pts). Otherwise, we computed the average of the two closest grades. It should be noted that in all cases where we chose two grades of the three, these two grades were always very close to each other.

E. Subjects

The subjects in this experiment are 18 3rd year students at the computer science department of Netanya Academic College, and two faculty members. In total we had 20 subjects. All participants except three were males. The average age is 24.8 (SD=8.7), and subjects are without industrial experience except one subject who had 3 years experience before his academic studies.

To ensure fair comparisons we asked the subjects about their average grades in general and in programming courses. Initially assignment was random, but later we assigned subjects to groups so as to reduce the variability in grades. Table II shows the averages of the 4 groups. According to this table we see that in terms of groups and style the averages are quiet similar.

F. Procedure

The first author was the experimenter of all subjects. The experimenter initially gave a general overview about the experiment and the eye tracker. Participants were told that the experiment is about comprehension but were not told the specific goal. The experimenter showed each participant how the eye tracker operates and let him practice that by himself. In particular, the experimenter asked each participant to notice the track-status window that shows the subject's eyes and their gazes. This is important because when the participant moves his head it is reflected in this window allowing the participant to learn about the valid range of his head's movements.

Once the participant felt satisfied with the system, the experimenter asked him to calibrate. The system notification about the calibration results uses a five-level scale. Table III shows the different levels, their accuracy, and the number of subjects at each level. The subject who failed the calibration process was tracked manually (he was requested to move the mouse to show the code he is looking at). Luckily he was assigned to a non-regular function, so was not needed for the detailed analysis of regular ones.

After the calibration phase the subject started the experimentation. The first screen presents a general overview and instructions, and the second screen presents the program to comprehend. The participant is allowed to study the program as much time as he wants and then answers the question. While studying the program he is allowed to use off-computer means to trace the variables even if this forces him to disconnect his gaze from screen.

A post-experiment question was asked by the experimenter about the way the subject studied the program. The initial question was "how did you approach the program". In the ensuing discussion subjects were also asked where they invested effort. They were also shown the heatmap of their gazes trying to learn more about the process, and asked to comment on it — specifically, whether it reflects what they think they did.

G. Study Variables

The dependent variables of this study are *correctness*, *completion time*, and *visual effort*. The *correctness* variable is the score a subject achieves for answering the *what does the function do?* question. The *completion time* variable measures the time a subject spent in the function stimuli including answering the question. The rationale of considering the time of writing the answers is that subjects also consider the stimuli while writing their answers.

The *correctness* and *completion time* variables are not the main variables we want to analyze in this study as they have been studied already in a previous work for comparing the comprehension of regular and non-regular implementations of the same program. Thus we use them for replication and for generating a challenging environment to get a realistic measure for the *visual effort* variable.

The *visual effort* variable measures, in terms of eye movements, the efforts a subject needs to get an answer. It is a *latent* variable so it is measured indirectly using *observable* variables related to fixations.

Fixation is one of two types of data that are considered when using the eye tracking technique. It occurs when the eyes stabilize on an object. The other type of data is called *saccade*. It describes a rapid movements between fixations.

We derive our observable variables from fixations rather than saccades as two important mental activities occur during fixation. These activities are derived from two assumptions that relate fixation to comprehension. The *eye-mind* assumption states that processing occurs during fixation, and the

immediacy assumption posits that interpretation at all levels of processing are not deferred [12].

The observable variables that are measured to represent visual effort are *fixation count*, *total fixation time*, and *pupil dilation*.

1) Fixation Count: This metric counts the number of fixations in a predefined area of interest (AOI).

2) Total Fixation Time: This metric measures the total fixation durations in a predefined AOI.

3) Average Pupil Dilation: It has been shown that there is a positive correlation between cognitive effort and pupil size. Hess showed that the pupil size increases with the increase of arithmetic complexity [8]. In a different domain, Just et al. found a correlation between pupil size and sentence complexity during a comprehension task [13]. Similar findings were presented by Ganholm et al. in the context of working memory load [6].

On the basis of the above works we use pupil dilation as a measure of complexity and apply it to compare regular code with non regular code. In addition, we use pupil size to examine our claim about decreasing complexity of repeated instances of code segments.

We define the metric as the average of pupil size in all fixations in a given area of interest (AOI). We discard gazes between fixations (saccads) as pupil size reflects complexity and complexity is experienced during processing which happens in fixations.

IV. Results and Analysis

A. Regular vs. Non-Regular Versions

1) Correctness and Time: The main purpose of this work is exploring the way developers approach regular code rather than comparing its performance to non regular code. We have investigated this in a previous work [10]. Nevertheless, we replicate that work to confirm the results. We use the following hypotheses to test the differences between regular and non-regular versions of the same program.

- H_0: Programmers achieve similar scores and time in understanding non-regular versions as in the regular counterpart.
- H_1: The regular versions are easier and faster to understand even though they are longer and have higher values of McCabe's cyclomatic complexity.

To test our hypotheses we initially look at the means of all regular and non-regular scores for each program, then consider the whole distribution of regular scores against the whole distribution of non-regular ones.

The four groups' scores met the normality assumption which was tested by the Shapiro-Wilks test. The *diamond* groups did not meet the equality of variance assumption so we did not assume that. As the groups are unrelated we used the independent t-test. Comparing the means of the regular and non-regular groups of the *diamond* programs yielded a significant different between these two groups ($t(4.967) = -3.211, p = 0.012$). So we can reject the null hypothesis and accept the alternative one. When examining the groups of the *median* program the difference between the means was not significant. Thus we cannot reject the null hypothesis in this case.

TABLE IV
Correctness and completion time results for all implementations.

Style	Correctness average	Completion time average
Regular (median)	66.5±30.0	26.0±12.9
Regular (diamond)	92.0±9.8	25.7±12.3
	80.2±25.4	25.9±12.0
Non-regular (median)	65.0±28.7	25.2±7.4
Non-regular (diamond)	49.1±27.0	31.5±21.6
	56.1±26.7	28.7±16.3

One explanation for the similar scores in the median program is that the difference between the values of the regularity measure for the regular and non-regular versions is not large enough. Furthermore, the non-regular version contains a code segment that computes the median by partial sorting. As sorting is a programming plan [25] it might serve as a strong clue for the whole function understanding.

Taken together, the results show that the regular versions are not more difficult, contradicting the naive expectation that subjects of the regular version achieve lower scores due to high values of LOC and MCC.

We also compared the whole distribution of regular scores (of the two programs) to the distribution of non-regular scores. We did not use the independent t-test as the groups failed the normality assumption even under transformation. In such cases it is recommended to use the Mann-Whitney non-parametric test. This test is used to compare differences between two unrelated groups when their dependent variable is not normally distributed. By running this test it was found that the regular group achieved significantly better scores than the non-regular group ($U = 24, p = 0.028$).

In terms of completion time, we also applied the independent t-test as the four groups were normally distributed and each pair also met the *equality of variance* assumption. For the two programs there was no significant difference in the means, so we cannot reject the null hypothesis.

According to Table IV the results are quite similar for the two styles in the two programs (with slight advantage for the regular style despite its long implementations when compared to the non regular style), except for one non-regular implementation (*diamond* program) where one subject in this group spent much time and as a result the average got a relatively high value.

These results (correctness and completion time) follow those of our previous work where we used the same functions as in this work [10].

2) Difficulty of Programming Style: Besides the *what does the function do?* question, we also asked the subjects to rank the function difficulty on an ascending 5-point scale. Figure 1 shows the distribution of the subjects' answers. In particular it shows that a third of the subjects of the non-regular implementation ranked their functions as very hard

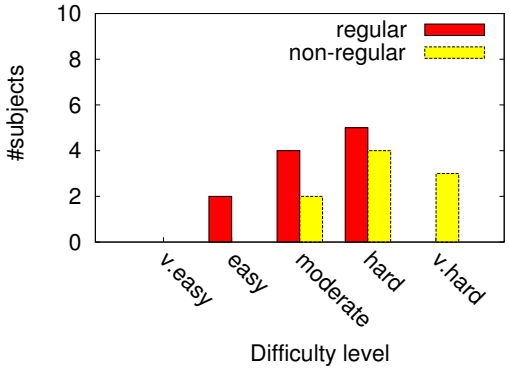


Fig. 1. Distribution of perceived difficulty ratings.

while none of the regular-implementation subjects used this level. On the opposite side, 2 subjects have ranked the regular implementations as easy while not even one subject of the non-regular group used this rank.

Moreover, about 55% of the regular group ranked their functions as easy or moderate while about 78% of the non-regular group ranked their functions as hard or very hard.

B. Visual Effort

1) Heat Map: One way to identify regions with special attention is using *heat maps*. They are designed to visualize fixations taken from many subjects. By this we can answer questions like *what locations of the stimulus are noticed by the average subject?* We use this technique to investigate whether subjects follow an obvious pattern in terms of effort allocation, and by this we answer our first research question.

Figure 2 shows the heat maps of the regular implementations (*diamond* and *median* programs). Both maps show that the average subject largely fixates on the first instance of the repeated pattern. The innermost red spot indicates the region that received the largest attention, and as we move downward the color becomes colder and regions get less attention. These figures show an aggregation of all subjects of each regular group.

The conclusion is that subjects spend more effort in the initial instances. When it comes to the last instances the examined area gets minimal focus.

Importantly, subjects did refocus on the final processing that comes after the regular repeated instances in the *median* program. This shows that attention is not just reduced with length, and subjects do not just tend to ignore the end of the function. Thus it strengthens the above result concerning reduced attention to repeated segments. The *diamond* program does not have such a final processing part.

There is no such obvious behavior in the non-regular counterparts as shown in Figure 3. Subjects generally focus on the inner-loop of the functions.

2) Areas of Interest: heat maps show the dominant areas in the code without clear separation between repeated segments. Areas of interest are geometric areas defined by the experimenter for the sake of between-area and within-area analyses.

TABLE V
MEASURES (AVERAGES) OF THE AOIS OF THE *median* REGULAR IMPLEMENTATION.

AOI	#fixations	Complete fixation time	Pupil size
AOI1	311.6	133714.0	19.98
AOI2	393.8	182459.3	19.85
AOI3	287.6	144096.0	19.54
AOI4	173.0	82537.0	19.55
AOI5	130.0	66345.5	19.38
AOI6	116.0	49318.6	19.19
AOI7	97.6	43115.3	19.11
AOI8	87.0	31101.8	19.18

TABLE VI
MEASURES (AVERAGES) OF THE AOIS OF THE *diamond* REGULAR IMPLEMENTATION.

AOI	#fixations	Complete fixation time	Pupil size
AOI1	496.7	235035.8	22.57
AOI2	239.0	92919.0	22.19
AOI3	179.0	80597.5	22.21
AOI4	129.2	46648.0	22.61

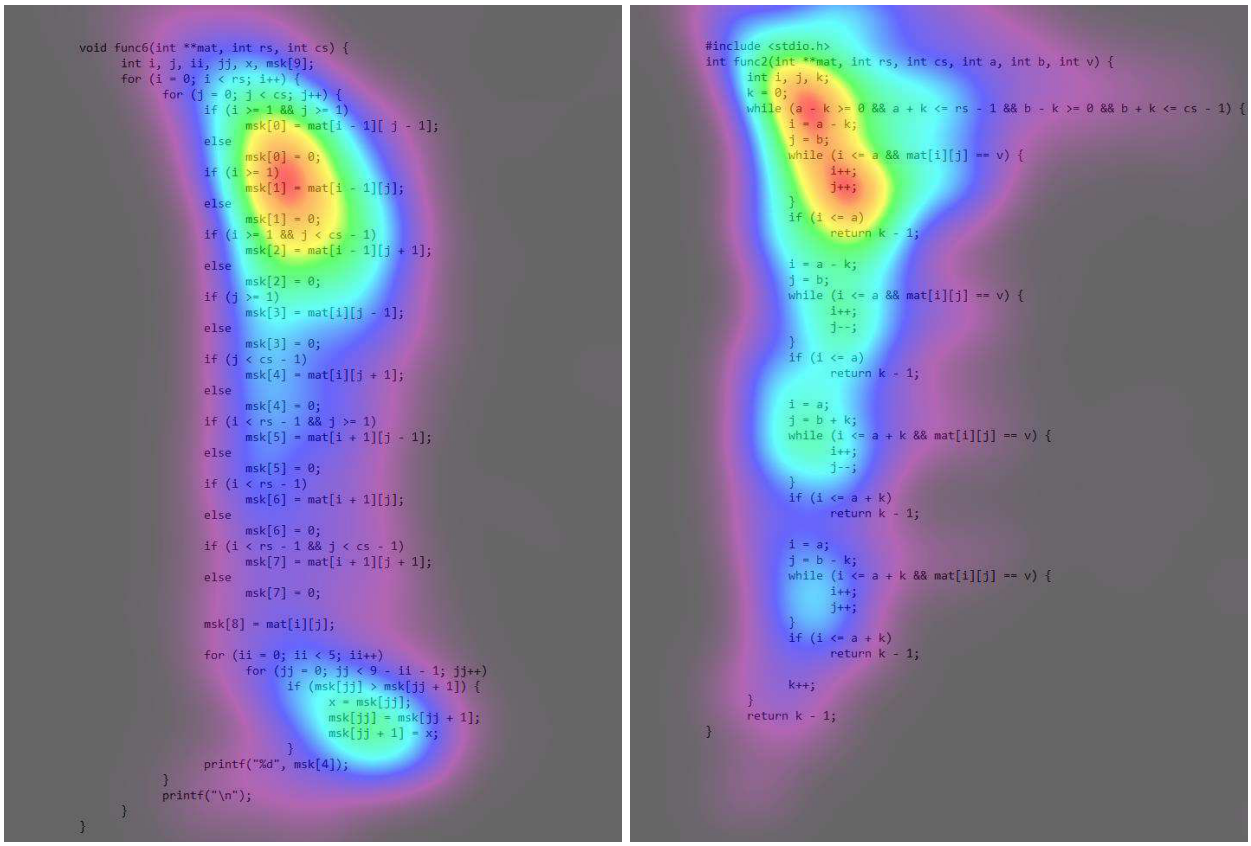
In both regular implementations we are interested in the repeated instances. The *median* version was divided into 8 areas of interest as shown in Figure 4 (one AOI for each instance), and the *diamond* version was likewise divided into 4 areas of interest (Figure 5). For each AOI we compute *fixation count*, *total fixations time*, and *average size of subjects' pupil*.

Tables V and VI show the measures of all areas of interest for the regular versions of both programs. Obviously these results show that subjects spent more time (and thus effort) in the earlier segments, and the time spent is sharply reduced as we progress to later segments. This behaviour is preserved in terms of all measures with slight digression in few cases, but the general pattern is pretty evident.

If subjects spend more time in one area rather than others that would normally mean that this area is more complex than others. But in our study, given that the segments are pretty similar, a better interpretation is that once one segment is learnt it is easier to comprehend the others.

3) Pupil Dilation: It is well known that there is a strong correlation between pupil size and mental effort, and therefore also a correlation between pupil size and the complexity of the task. Table V shows the average size of the subjects' pupil in all areas of interest in the regular version of the *median* program. According to this table the average size in the first AOI is 19.98, in the second AOI it is 19.85, in the third 19.54, and this behaviour is roughly preserved as we progress to the next areas. Similar behaviour occurs in the regular version of the *diamond* program (Table VI). Thus the pupil size data too indicates that successive repeated code segments become easier to comprehend.

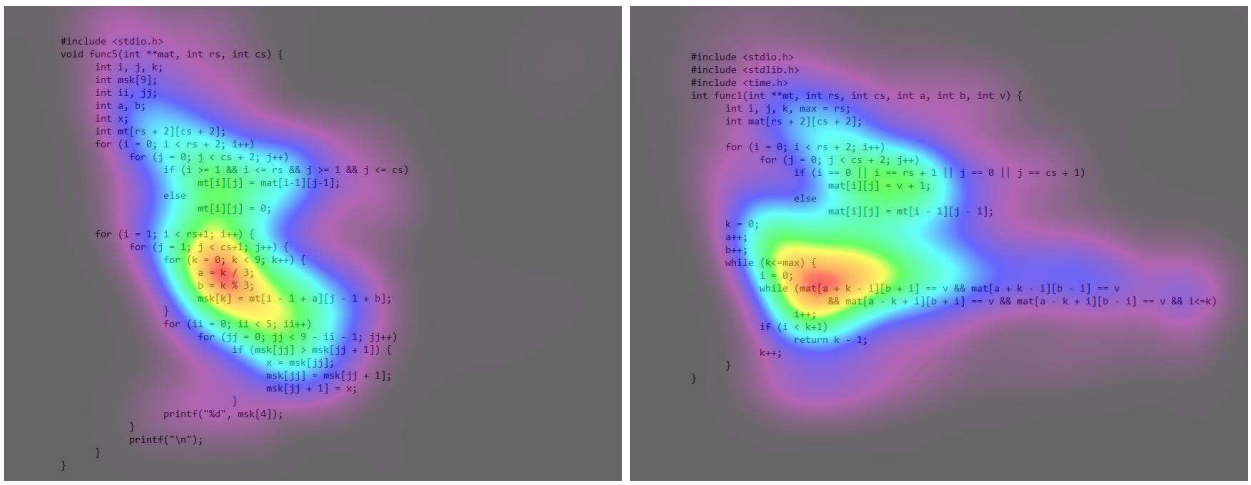
However, the differences are smaller than for the fixation data. To investigate this more deeply, we consider the individual distributions for the different subjects. As shown in Figure 6, some have a clear downward trend, e.g. *Subject5* and *Subject1*. For others there is a mainly downward trend, but it is not monotonous — as for *Subject2* and *Subject9*. Finally



```
void func6(int **mat, int rs, int cs) {
    int i, j, ii, jj, x, msk[9];
    for (i = 0; i < rs; i++) {
        for (j = 0; j < cs; j++) {
            if (i >= 1 && j >= 1)
                msk[0] = mat[i - 1][ j - 1];
            else
                msk[0] = 0;
            if (i >= 1)
                msk[1] = mat[i - 1][j];
            else
                msk[1] = 0;
            if (i >= 1 && j < cs - 1)
                msk[2] = mat[i - 1][j + 1];
            else
                msk[2] = 0;
            if (j >= 1)
                msk[3] = mat[i][j - 1];
            else
                msk[3] = 0;
            if (j < cs - 1)
                msk[4] = mat[i][j + 1];
            else
                msk[4] = 0;
            if (i < rs - 1 && j >= 1)
                msk[5] = mat[i + 1][j - 1];
            else
                msk[5] = 0;
            if (i < rs - 1)
                msk[6] = mat[i + 1][j];
            else
                msk[6] = 0;
            if (i < rs - 1 && j < cs - 1)
                msk[7] = mat[i + 1][j + 1];
            else
                msk[7] = 0;

            msk[8] = mat[i][j];

            for (ii = 0; ii < 5; ii++)
                for (jj = 0; jj < 9 - ii - 1; jj++)
                    if (msk[jj] > msk[jj + 1]) {
                        x = msk[jj];
                        msk[jj] = msk[jj + 1];
                        msk[jj + 1] = x;
                    }
            printf("%d", msk[4]);
        }
        printf("\n");
    }
}
```

```
#include <stdio.h>
int func2(int **mat, int rs, int cs, int a, int b, int v) {
    int i, j, k;
    k = 0;
    while (a - k >= 0 && a + k <= rs - 1 && b - k >= 0 && b + k <= cs - 1) {
        i = a - k;
        j = b;
        while (i <= a && mat[i][j] == v) {
            i++;
            j++;
        }
        if (i <= a)
            return k - 1;

        i = a - k;
        j = b;
        while (i <= a && mat[i][j] == v) {
            i++;
            j--;
        }
        if (i <= a)
            return k - 1;

        i = a;
        j = b + k;
        while (i <= a + k && mat[i][j] == v) {
            i++;
            j--;
        }
        if (i <= a + k)
            return k - 1;

        i = a;
        j = b - k;
        while (i <= a + k && mat[i][j] == v) {
            i++;
            j++;
        }
        if (i <= a + k)
            return k - 1;

        k++;
    }
    return k - 1;
}
```

Fig. 2. Left: heat map of the regular implementation of the *median* program based on 6 subjects. Right: heat map of the regular implementation of the *diamond* program based on 4 subjects (we excluded the fifth subject due to a contradiction between his heat map and think-aloud results).



```
#include <stdio.h>
void func5(int **mat, int rs, int cs) {
    int i, j, k;
    int msk[9];
    int ii, jj;
    int a, b;
    int x;
    int mt[rs + 2][cs + 2];
    for (i = 0; i < rs + 2; i++)
        for (j = 0; j < cs + 2; j++)
            if (i >= 1 && i <= rs && j >= 1 && j <= cs)
                mt[i][j] = mat[i-1][j-1];
            else
                mt[i][j] = 0;

    for (i = 1; i < rs+1; i++) {
        for (j = 1; j < cs+1; j++) {
            for (k = 0; k < 9; k++) {
                a = k / 3;
                b = k % 3;
                msk[k] = mt[i - 1 + a][j - 1 + b];
            }
            for (ii = 0; ii < 5; ii++)
                for (jj = 0; jj < 9 - ii - 1; jj++)
                    if (msk[jj] > msk[jj + 1]) {
                        x = msk[jj];
                        msk[jj] = msk[jj + 1];
                        msk[jj + 1] = x;
                    }
            printf("%d", msk[4]);
        }
        printf("\n");
    }
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int func1(int **mt, int rs, int cs, int a, int b, int v) {
    int i, j, k, max = rs;
    int mat[rs + 2][cs + 2];

    for (i = 0; i < rs + 2; i++)
        for (j = 0; j < cs + 2; j++)
            if (i == 0 || i == rs + 1 || j == 0 || j == cs + 1)
                mat[i][j] = v + 1;
            else
                mat[i][j] = mt[i - 1][j - 1];
    k = 0;
    a++;
    b++;
    while (k<=max) {
        i = 0;
        while (mat[a + k - i][b + i] == v && mat[a + k - i][b - i] == v
                && mat[a - k + i][b + i] == v && mat[a - k + i][b - i] == v && i<=k)
            i++;
        if (i < k+1)
            return k - 1;
        k++;
    }
}
```

Fig. 3. Left: heat map of the non-regular implementation of the *median* program based on 3 subjects (we excluded the fourth subject as he failed the calibration process). Right: heat map of the non-regular implementation of the *diamond* program based on 5 subjects.

there are those where there is no clear trend, and even one with an upward trend. In general, the subjects who worked on the *median* program exhibited stronger trends, maybe because the *diamond* program had only 4 repeated segments.

4) Verification of Eye Gaze Results: A post-experiment question was asked by the experimenter of each of the subjects about their approach and effort allocation to the different parts of the function. During the conversation they were presented with the heat map of their session and were asked whether this map matches their subjective impression. In particular the focus was on the subjects of the regular implementations. We summarize their responses in Table VII. According to this table more than 72% of the subjects stated clearly that they spent more time on the first instances. One subject just stated that

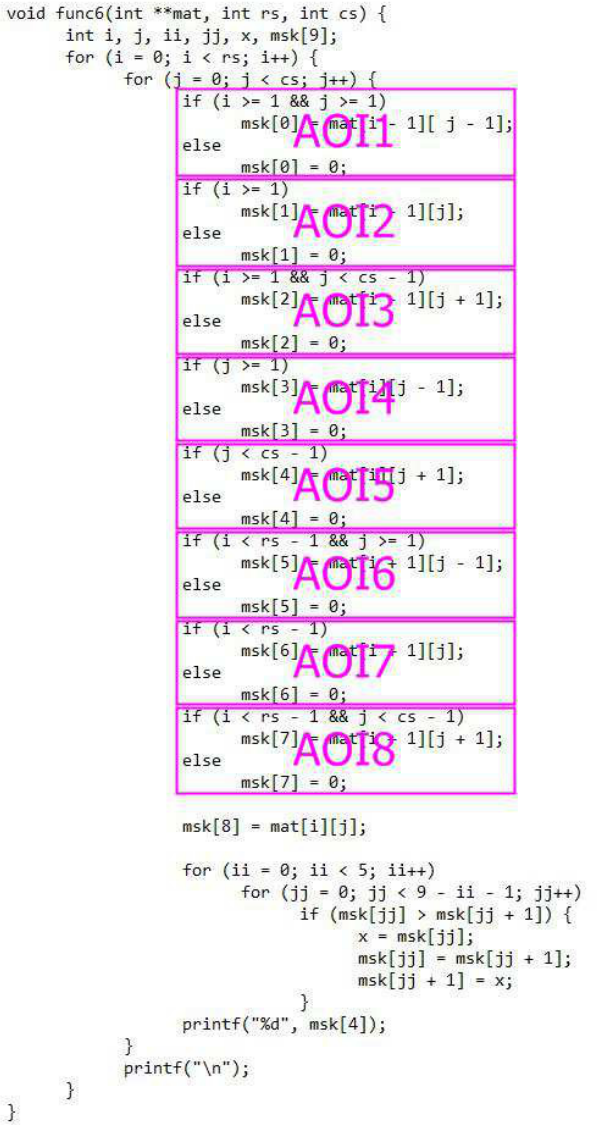


Fig. 4. The areas of interest (AOIs) of the *median* regular implementation.

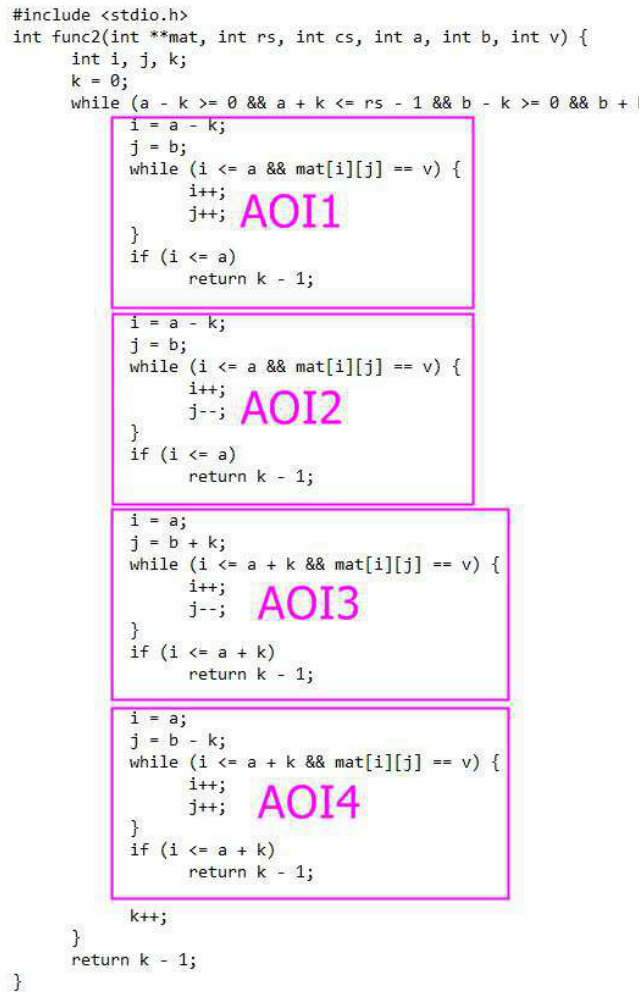


Fig. 5. The areas of interest (AOIs) of the *diamond* regular implementation.

instances are similar without any statement regarding effort allocation. Two subjects did not express awareness of the regularity issue.

The responses of *Subject20* and *Subject19* were particularly interesting. *Subject20* did not agree with his heat map and said that he did not investigate the program this way. His heat map shows one spot on the last inner while and one before the outermost loop. We believe that something went wrong while recording the gazes. It could be that the device was unintentionally moved by the subject or the subject himself moved.

Subject19 was surprised from the perfect matching between her mind and its heat map. She was even more surprised when she realized that her pattern follows the aggregated pattern of all other subjects. She said that she always thinks differently and it is interesting to see that this time she broke that.

C. Modeling Effort in Repeated Instances

We claim that not all code segments in a program should have equal weight, especially if they have the same structure or they are clones. The rationale is that once the developer understands one instance it is easier for him to understand the other instances and therefore he needs less effort.

Based on this claim we observe that many widely used complexity metrics unfairly present inflated measurements of a given code. For example, the McCabe cyclomatic complexity is based on the number of conditions in the code where all conditions are treated the same. Conditions in the 10th instance of a pattern are counted just like those in the first instance. But this is misleading. As we showed, developers do not need to invest the same effort in repeated instances.

Our purpose is therefore to build a model that predicts efforts needed to understand a repeated instance on the basis of its ordinal number. To do so we use the fixation data for all the subjects and check the fit of candidate functions to this data. The natural candidates are various decreasing functions. Table VIII shows the models found by the curve

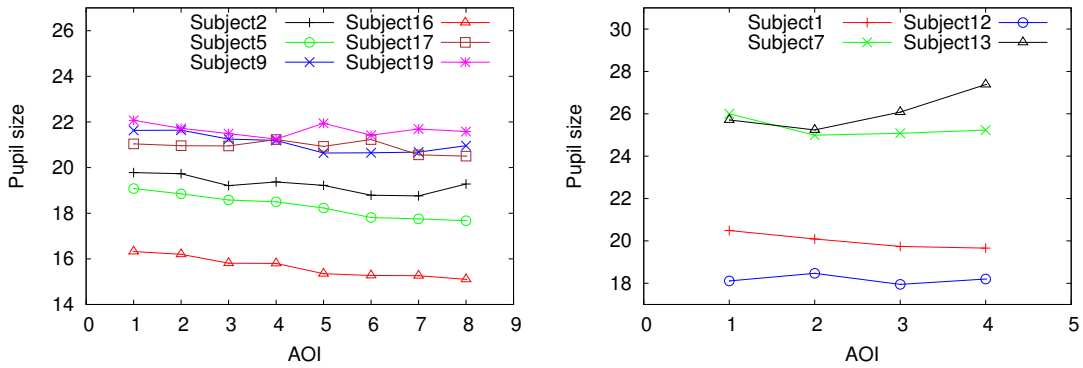


Fig. 6. Left: Changes to the pupil size of 6 subjects over 8 AOIs of the regular version of the *median* program. Right: Changes to the pupil size of 4 subjects over 4 AOIs of the regular version of the *diamond* program.

TABLE VII
SUBJECT OPINIONS REGARDING THEIR EFFORT ALLOCATIONS IN THE REGULAR IMPLEMENTATIONS.

Subject Version	Response
Subject1 diamond	I realized that once I understood the first segment, it will be easier to understand the rest due to similarity.
Subject2 median	Do not know why there is more focus on the first segment compared to others.
Subject5 median	Passed over all the code but focused on the first *if* more than others. I saw that the segments are similar so spent less efforts in the later. If the later segments were different I would spent more effort there.
Subject7 diamond	Inner loops were similar.
Subject9 median	Passed over all ifs, but it was enough to focus on a few to understand others.
Subject12 diamond	Spent much efforts at the beginning, tried to understand the loops at the beginning because I saw that they repeat themselves. In particular I realized that the differences are very small so it is easy to infer about other.
Subject13 diamond	Spent more efforts on the first inner loop because it is new for me and the rest are similar.
Subject16 median	Passed over all loops and ifs. Spent much efforts on the ifs.
Subject17 median	Most of the time in the ifs. Thought about one *if* and infer about others.
Subject19 median	I was panicked of the *if...else* statements but once saw they all similar I spent much time on those at the beginning. She was surprised from the fact that her attention map follows the pattern of the other and said that she always thinks in a different way than others.
Subject20 diamond	Most of the efforts were spent on the inner loops in particular the first one because it "jumps to the eyes" the similarity with others. I do not agree with the heat map (it shows he spent much efforts on the last loop), it does not reflect the real efforts I spent.

fitting procedure for the different measures (complete fixation time and number of fixations) as a function of AOI for our two regular implementations. According to the table all the models are significant except one (the exponential model of the *complete fixation time* measure of the *diamond* program).

The best model turns out to depend on the program. For the *median* program the best model could be the *exponential* one, which explains about 46–49% of the observed variation. Other models are far behind it. As for the *diamond* program the best model is the *cubic* one. It succeeds in explaining more than 51% of the observed variation in both measures. An additional good candidate is the *quadratic* model which is relatively close to the *cubic* model. The worst model for this program is the *exponential*, which in one case is not statistically significant and in the other explains only 30% of the variation which is relatively low.

The reason for the relatively low values of the R^2 of the different models is the way the observed values distribute. For example, for the *median* version we have 8 AOIs. For each AOI we have a column of the measurements for each subject. Due to the natural variability between subjects, it is impossible to explain all the variation using a function of only the instances.

But as we are interested in the *average user* on the long term we can perhaps do better if we fit a model to the average value for each AOI. Thus the data is reduced to a single vector with 8 values for each measure in the *median* program, and 4 values for the *diamond* program. In fact these values are the ones shown in Tables VI and V.

The results of fitting the *median* program data are that all model equations are pretty much similar (up to fractional digits) to those of Table VIII and statistically significant. The substantial change was in their R^2 values. In particular, the *exponential* model explained 92% of the variation while the worst model explained a bit more than 70%.

As for the *diamond* program the results show that the *linear* and *quadratic* models are not significant, so we are left with the other three. Of these, the *cubic* model shows a perfect fit, while the other two show a very high fit. However, note that with only 4 data points a cubic function can indeed pass through all the points, so this may be an overfit.

Indeed, when selecting a model one should consider the characteristics of the function and not only the R^2 of the fit. For example, the five model functions for the number of fixations on the *diamond* program from Table VIII are shown in Figure 7 (right). This shows that as we extrapolate to larger xs, the quadratic model grows to infinity, while the logarithmic, linear, and cubic models attain negative values.

TABLE VIII
RESULTS OF CURVE FITTING TO FIXATION DATA AS A FUNCTION OF INSTANCE NUMBER IN REGULAR IMPLEMENTATIONS.

Version	Measure	Equation	Model	Sig.	R^2
median	complete fixation time	Linear	$y = -20422.9x + 183489.2$	0	0.340
		Logarithmic	$y = -65923.5 * ln(x) + 178972.5$	0	0.292
		Quadratic	$y = 325.1x^2 - 23349.5x + 188366.8$	0	0.340
		Cubic	$y = 1743.5x^3 - 23212.9^2 + 66443.9x + 102060.5$	0	0.375
		Exponential	$ln(y) = -0.254x + 12.2$	0	0.465
	#fixations	Linear	$y = -43x + 393.1$	0	0.384
		Logarithmic	$y = -144.3 * ln(x) + 390.9$	0	0.356
		Quadratic	$y = 3.2x^2 - 72.6x + 442.6$	0	0.393
		Cubic	$y = 3.1x^3 - 38.9x^2 + 88.6x + 287.5$	0	0.422
		Exponential	$ln(y) = -0.228x + 6.0$	0	0.489
diamond	complete fixation time	Linear	$y = -57748.2x + 258171$	0.007	0.414
		Logarithmic	$y = -132855.2 * ln(x) + 219355.4$	0.003	0.475
		Quadratic	$y = 27041x^2 - 192958x + 393380.6$	0.013	0.487
		Cubic	$y = -25237x^3 + 216320.3x^2 - 614418x + 658370.5$	0.029	0.515
		Exponential	$ln(y) = -0.405x + 12.2$	0.066	0.222
	#fixation	Linear	$y = -116.2x + 551.6$	0.006	0.432
		Logarithmic	$y = -265.6 * ln(x) + 472.0$	0.003	0.489
		Quadratic	$y = 52x^2 - 376.2x + 811.6$	0.011	0.501
		Cubic	$y = -31.2x^3 + 286.3x^2 - 898.1x + 1139.7$	0.03	0.512
		Exponential	$ln(y) = -0.375x + 6.2$	0.026	0.308

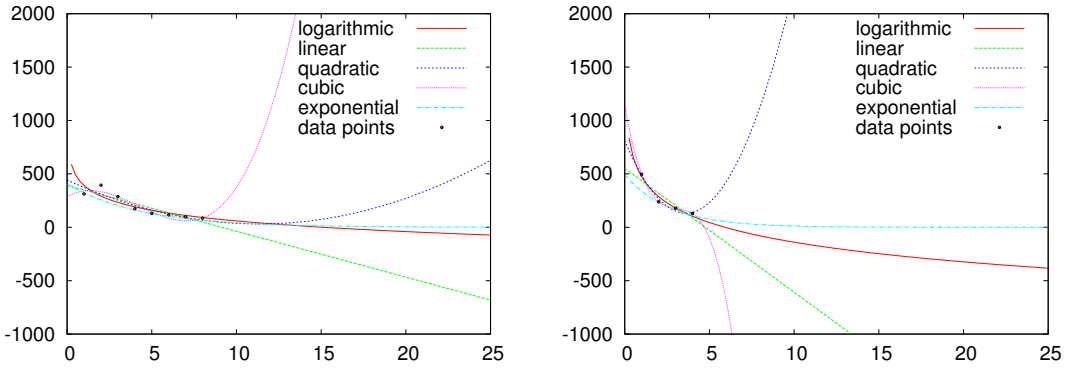


Fig. 7. Extrapolation of the model functions (of the #fixation measure) from Table VIII. Data points are from Table V. Left: *median* version. Right: *diamond* version.

The exponential model has the more appropriate attribute of tending asymptotically to zero.

As for the *median* version (Figure 7 left) the linear, logarithmic, and quadratic models behave as in the *diamond* program although not as steeply. However, the cubic model changed its direction to positive infinity to compensate for the non-monotonicity of the first point.

On the basis of the above observations one may claim that all models, except the exponential one, are bad for extrapolation as some of them grow to infinity and others to negative values for larger xs. Theoretically that is right, however, the number of repeated instances in the code does not grow to very large values. Therefore, for some thresholds other models could be a good fit.

V. THREATS TO VALIDITY

The results of this work are subject to several threats to validity, in particular in the experimental part.

There is an obvious advantage to using a remote eye tracker over a head mounted device, especially when considering intrusiveness and how natural is the experiment environment. Yet, it is still somewhat restrictive and may influence subjects' behavior and affect their performance. For example, one subject noted a fear to move his head too much which prevented him from fully tracing the function.

The small number of subjects in each group is another threat to validity. It is hard to avoid because of the need to conduct personal experiments with the eye tracker, and our total of 20 is relatively high in this context when compared to other works that use eye tacking [1], [21], [22], [30].

In this work we only used two different programs and our conclusions rely on them. The hope is to generalize to additional examples. The reason we stuck to these programs is because we already used them in our previous work, and they appear to be non-trivial and realistic. Furthermore, this work basically uses undergraduate students which could limit its generalization.

Two more threats are related to the areas of interest (AOIs). In our analysis each area of interest captures one repeated instance. However, repeated instances may form a continuum, therefore, areas of interest may span over two successive

instances. Moreover, we used the same margins around the code of each instance, and created rectangular areas, but other options and geometric shapes are possible and may lead to slightly different results.

VI. Related Work

A large body of work has been done in the area of syntactic complexity metrics. Lines of code (LOC) is a very straightforward metric that simply counts lines. Halstead defined the software science metrics including one which measures programming effort [7]. This is built on the basis of operator and operand occurrences. McCabe introduced the cyclomatic complexity metric which simply counts the number of conditions in the code [15].

These metrics and others simply counts syntactic elements. But are all lines in the code of equal importance? Do all operators or operands have the same effect? Do all constructs and conditions have the same intrinsic complexity? A few works have considered these questions and introduced weight-based metrics. For example, the cognitive functional size (CFS) metric is based on cognitive weights of the different control structure [19]. Oman et al. defined the maintainability index on the basis of three other syntactic metrics [17], [28].

Admittedly, these works have taken the syntactic metrics one step forward, but they still ignore the *context* of source code elements. In particular, repeated structures are based on the same elements but require different cognitive effort for the comprehension process. As far as we know we are the first to empirically quantify the effect of context on complexity as anticipated by Weyuker [29].

There have been other works that study repetitions in code. Vinju et al. empirically showed that the cyclomatic complexity metric overestimates understandability of Java methods. They introduced *compressed control flow patterns* (CCFPs) that summarizes consecutive repetitive control flow structure, which helps in identifying where and how many times the cyclomatic metric overestimates the complexity of the code [26]. But their focus was not on complexity or regularity, but rather on the question of whether people understand control flow by recognizing patterns. Nevertheless, in the analysis they assert that "code that looks regular is easier to chunk and therefore easier to understand".

Sasaki et al. were even closer to our work. They recognized that one reason for large values of the MCC metric is the presence of consecutive repeated structures, and suggested that humans would not have difficulty in understanding such a source code. They then proposed performing preprocessing to simplify repeated structures for metrics measurement [18]. But both these works lack quantitative experimental evidence, and we are not aware of such evidence also in the context of clones in source code.

Eye tracking has recently been used in several code comprehension studies. Sharif et al. have used eye tracking in multiple works. In [21] eye tracking was used to capture quantitative data to investigate the affect of identifier-naming conventions on code comprehension. The use of eye tracking was a better alternative to traditional means that were used in a previous similar work [2]. Likewise, in [22] they also replicate a previous work where traditional means were used. The replication uses eye tracking to extend the results and determine the effect of layout on the detection of roles in design patterns. Finally, in [20] they replicated a previous eye tracking study, but with more participant and additional eye-tracking measures. This work investigates how programmers find defects in source code.

Yusuf et al. used eye tracking to identify the most effective characteristics of UML class diagrams that support software tasks.

As for program comprehension measurement, there have been works that use psycho-physiological sensors and functional magnetic resonance imaging [24], [5].

We know of no previous work that used eye tracking to quantify complexity model parameters.

VII. Conclusions

We conducted an eye tracking experiment to see how programmers read code when they try to understand it, for regular and non-regular versions of the same programs. Results show that in the repeated segments the programmers tend to invest more effort on the initial repetitions, and less and less on successive ones. Specifically, the time and number of fixations seem to drop of exponentially (although other models, e.g. cubic, are also possible).

One may claim that the fact that programmers invest less effort in the later repeated instances is a natural behaviour which stems from fatigue or lack of interest. However the heat map of the *median* version showed that subjects renewed focus on the last segment of the function which is not part of the repetitive segments. Thus we can claim that the reduced attention is indeed a function of the repetitions.

The reduced attention is related to the fact that repeated patterns can be anticipated and are easier to understand, as was verified by post-experiment debriefing with participants. The above observations therefore indicate that syntactic complexity metrics, which just count the number of appearances of various syntactic constructs, should be modified with context-dependent weights. For example, assuming an exponential model with a base of 2, a modified version of the MCC metric would add the full MCC of the first instance, but only $\frac{1}{2^{i-1}}$ of the MCC of the ith instance. This shows how syntactic measures can be reconciled with Weyuker's suggestion that complexity metrics reflect context [29].

However, the current experiments are not extensive enough to enable a full model to be formulated. Of the two programs we used, one produced results which favor an exponential model, while the other's results do not. Additional measurement with more programs and subjects are needed in order to converge on a general model, or alternatively, to identify when different models are appropriate.

Acknowledgments

This research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 407/13).

REFERENCES

[1] R. Bednarik and M. Tukiainen, "*An eye-tracking methodology for characterizing program comprehension processes*". In *Proceedings of the 2006 Symposium on Eye Tracking Research & Applications*, pp. 125–132, ACM, New York, NY, USA, 2006, ISBN 1-59593-305-0, doi: 10.1145/1117309.1117356.

[2] D. Binkley, M. Davis, D. Lawrie, and C. Morrell, "*To camelCase or under_score*". In *IEEE 17th International Conference on Program Comprehension*, pp. 158–167, May 2009, doi:10.1109/ICPC.2009.5090039.

[3] B. Cornelissen, A. Zaidman, and A. van Deursen, "*A controlled experiment for program comprehension through trace visualization*". *IEEE Trans. Softw. Eng.* **37(3)**, pp. 341 –355, May-June 2011, doi: 10.1109/TSE.2010.47.

[4] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*. Course Technology, 2nd ed., 1998.

[5] T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, and M. Züger, "*Using psycho-physiological measures to assess task difficulty in software development*". In *Proceedings of the 36th International Conference on Software Engineering*, pp. 402–413, ACM, New York, NY, USA, 2014, ISBN 978-1-4503-2756-5, doi:10.1145/2568225.2568266.

[6] E. Granholm, S. K. Morris, A. J. Sarkin, R. F. Asarnow, and D. V. Jeste, "*Pupillary responses index overload of working memory resources in schizophrenia*". *Journal of Abnormal Psychology* **106**, 1997.

[7] M. Halstead, *Elements of Software Science*. Elsevier Science Inc., 1977.

[8] E. H. Hess, "*Attitude and pupil size*". *Scientific American* **212**, 1965.

[9] A. Jbara and D. G. Feitelson, "*Quantification of code regularity using preprocessing and compression*". Manuscript, Jan 2014.

[10] A. Jbara and D. G. Feitelson, "*On the effect of code regularity on comprehension*". In *Proceedings of the 22nd International Conference on Program Comprehension*, pp. 189–200, ACM, New York, NY, USA, 2014, ISBN 978-1-4503-2879-1, doi:10.1145/2597008.2597140.

[11] A. Jbara, A. Matan, and D. Feitelson, "*High-MCC functions in the Linux kernel*". *Empirical Software Engineering* **19(5)**, pp. 1261–1298, 2014, doi:10.1007/s10664-013-9275-7.

[12] M. Just and P. Carpenter, "*A theory of reading: From eye fixations to comprehension*". *Psychological Review* **87**, pp. 329–354, 1980.

[13] M. A. Just and P. A. Carpenter, "*The intensity dimension of thought: Pupillometric indices of sentence processing*". *Canadian Journal of Experimental Psychology* **47**, 1993.

[14] J. L. Krein, L. Pratt, A. Swenson, A. MacLean, C. D. Knutson, and D. Eggett, "*Design patterns in software maintenance: An experiment replication at Brigham Young University*". In 2nd *Intl. Workshop Replication in Empirical Software Engineering Research*, pp. 25–34, 2011, doi:10.1109/RESER.2011.10.

[15] T. McCabe, "*A complexity measure*". *IEEE Trans. Softw. Eng.* **SE-2(4)**, pp. 308–320, Dec 1976, doi:10.1109/TSE.1976.233837.

[16] N. Nagappan, T. Ball, and A. Zeller, "*Mining metrics to predict component failures*". In 28th *Intl. Conf. Softw. Eng.*, pp. 452–461, May 2006, doi:10.1145/1134285.1134349.

[17] P. Oman and J. Hagemeister, "*Construction and testing of polynomials predicting software maintainability*". *J. Syst. & Softw.* **24(3)**, pp. 251–266, Mar 1994, doi:10.1016/0164-1212(94)90067-1.

[18] Y. Sasaki, T. Ishihara, K. Hotta, H. Hata, Y. Higo, H. Igaki, and S. Kusumoto, "*Preprocessing of metrics measurement based on simplifying program structures*". In *19th Asia-Pacific Software Engineering Conference (APSEC)*, vol. 2, pp. 120–127, 2012, doi: 10.1109/APSEC.2012.59.

[19] J. Shao and Y. Wang, "*A new measure of software complexity based on cognitive weights*". *Canadian J. Electrical and Comput. Eng.* **28(2)**, pp. 69 –74, april 2003, doi:10.1109/CJECE.2003.1532511.

[20] B. Sharif, M. Falcone, and J. I. Maletic, "*An eye-tracking study on the role of scan time in finding source code defects*". In *Proceedings of the Symposium on Eye Tracking Research and Applications*, pp. 381–384, ACM, New York, NY, USA, 2012, ISBN 978-1-4503-1221-9, doi: 10.1145/2168556.2168642.

[21] B. Sharif and J. Maletic, "*An eye tracking study on camelCase and under_score identifier styles*". In *IEEE 18th International Conference on Program Comprehension (ICPC)*, pp. 196–205, June 2010, doi: 10.1109/ICPC.2010.41.

[22] B. Sharif and J. Maletic, "*An eye tracking study on the effects of layout in understanding the role of design patterns*". In *IEEE International Conference on Software Maintenance (ICSM)*, pp. 1–10, Sept 2010, doi: 10.1109/ICSM.2010.5609582.

[23] B. Shneiderman, "*Measuring computer program quality and comprehension*". *Intl. J. Man-Machine Studies* **9(4)**, July 1977.

[24] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann, "*Understanding understanding source code with functional magnetic resonance imaging*". In *Proceedings of the 36th International Conference on Software Engineering*, pp. 378–389, ACM, New York, NY, USA, 2014, ISBN 978-1-4503-2756-5, doi: 10.1145/2568225.2568252.

[25] E. Soloway and K. Ehrlich, "*Empirical studies of programming knowledge*". *IEEE Trans. Softw. Eng.* **SE-10(5)**, pp. 595–609, Sep 1984, doi: 10.1109/TSE.1984.5010283.

[26] J. J. Vinju and M. W. Godfrey, "*What does control flow really look like? Eyeballing the cyclomatic complexity metric*". In 12th *Working Conf. Source Code Analysis and Manipulation*, Sep 2012.

[27] H. Wang, M. Chignell, and M. Ishizuka, "*Empathic tutoring software agents using real-time eye tracking*". In *Proceedings of the 2006 Symposium on Eye Tracking Research & Applications*, pp. 73–78, ACM, New York, NY, USA, 2006, ISBN 1-59593-305-0, doi: 10.1145/1117309.1117346.

[28] K. D. Welker, P. W. Oman, and G. G. Atkinson, "*Development and application of an automated source code maintainability index*". *J. Softw. Maintenance* **9(3)**, pp. 127–159, May 1997.

[29] E. J. Weyuker, "*Evaluating software complexity measures*". *IEEE Trans. Softw. Eng.* **14(9)**, pp. 1357–1365, Sep 1988, doi:10.1109/32.6178.

[30] S. Yusuf, H. Kagdi, and J. Maletic, "*Assessing the comprehension of UML class diagrams via eye tracking*". In *15th IEEE International Conference on Program Comprehension*, pp. 113–122, June 2007, doi: 10.1109/ICPC.2007.10.